Marquise: Creating Complete User Interfaces by Demonstration

Brad A. Myers *Richard G. McDaniel* *David S. Kosbie*

School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213
{bam, richm, koz}@cs.cmu.edu

ABSTRACT

Marquise is a new interactive tool that allows virtually all of the user interfaces of graphical editors to be created by demonstration without programming. A ‘‘graphical editor’’ allows the user to create and manipulate graphical objects with a mouse. This is a very large class of programs and includes drawing programs like MacDraw, graph layout editors like MacProject, visual language editors, and many CAD/CAM programs. The primary innovation in Marquise is to allow the designer to demonstrate the overall behavior of the interface. To implement this, the Marquise framework contains knowledge about palettes for creating and specifying properties of objects, and about operations such as selecting, moving, and deleting objects. The interactive tool uses the framework to allow the designer to demonstrate most of the end user’s actions without programming, which means that Marquise can be used by non-programmers.

KEYWORDS: User Interface Software, User Interface Management Systems, Interface Builders, Demonstrational Interfaces, Garnet.

INTRODUCTION

One important goal of the Garnet project [6] is to allow user interface designers who are not programmers to design and implement the *look and feel* of user interfaces. The Marquise tool is the newest addition to the Garnet environment, and it ties together all the previous tools, while supporting, for the first time, interactive specification of the entire user interface.

In particular, Marquise allows the overall graphical appearance of the interface to be drawn, and the behaviors for object creation, selection and manipulation to be demonstrated.

Unlike many previous tools which concentrate on widgets, Marquise is aimed mostly at the main drawing window of graphical editors where the user creates and manipulates graphical objects with a mouse. For example, with Marquise you can demonstrate how the rubber banding will appear as you move the mouse, rather than having this as a hard-wired, unchangeable component. Another important capability in Marquise is demonstrating the modes of the interface. Although ‘‘mode-free’’ interfaces are often touted, all modern graphical interfaces are in fact highly moded. For example, in most drawing tools such as Macintosh MacDraw, a palette controls whether the next mouse click will select an object, insert a text string, or draw a rectangle, circle, polygon, etc. Other modes include the current colors, line styles, and arrowhead styles for the objects that will be created. Marquise provides an intuitive, demonstrational method for specifying the modes that control and are affected by an operation.

With Marquise, we have concentrated on providing complete control of when and how the behaviors are initiated. The primary innovations in Marquise are: (1) the use of special icons to represent the mouse positions while demonstrating the behavior, so the designer can then demonstrate what happens at those locations, (2) sophisticated control over the locations where those events should take place to begin and end behaviors, (3) a ‘‘mode window’’ to make explicit the modes of the interface that control the behaviors and values, (4) the formalization of ‘‘palettes’’ to control modes and object properties, and (5) the ability to interactively specify the attributes for built-in layout operations and objects.

Marquise stands for <u>M</u>ostly <u>A</u>utomated, <u>R</u>emarkably <u>Q</u>uick <u>U</u>ser <u>I</u>nterface <u>S</u>oftware <u>E</u>nvironment. (A ‘‘marquise’’ is a gem having the shape of a short, pointed oval with many facets.) Marquise is part of the Garnet system, which is a comprehensive user interface development environment written in Lisp for the X window system.[1]

[1]The Garnet system is available by anonymous FTP. Although Marquise is not yet ready for distribution as this paper is being written, you can get the toolkit, the Gilt interface builder, and Lapidary. Send mail to `Garnet@cs.cmu.edu` for information.

RELATED WORK

Previous design tools have shown that it is possible to interactively specify the graphical appearance and behavior of limited parts of an application's user interface. For example, many interface builders, such as the NeXT Interface Builder, UIMX for Motif, Druid [8], and Gilt [7], allow the designer to interactively specify the placement of widgets. Peridot [3] allows new widgets to be created interactively without programming, and Lapidary [4] allows application-specific graphical objects to be demonstrated. Marquise goes beyond these tools since it supports creating, editing, and deleting of objects at run time, and allows the overall behavior to be defined. DEMO [10] used the idea of demonstrating the end-user's actions that start a behavior (called the ''stimulus'') and then demonstrating the response to that stimulus. DEMO II [1] added sophisticated techniques for inferring constraints to control how objects are placed or moved. Marquise uses the stimulus-response idea, here called ''train'' and ''show,'' but concentrates on which high-level actions are appropriate and the context of the stimulus.

Some previous systems have provided frameworks to help code graphical editors. Unidraw [9] and many graph editors (e.g., [2]) provide a standard set of built-in operations as methods, and the designer writes code to override these methods for the specific application. However, none of these other systems allow new behaviors to be defined by demonstration.

USER INTERFACE

The basic windows for Marquise are shown in Figure 1. There is a palette of objects that can be drawn, some palettes for controlling the properties of those objects, and a set of commands in a menubar. In all conventional interface builders there are two modes: *Build* and *Run*, where in Build mode the designer constructs the interface, and in Run mode it is tested. Marquise adds two additional modes to demonstrate behaviors: *Train* and *Show*. Train mode is used to demonstrate what the end user will do, and Show mode is used to demonstrate the system's response to that action. A different mouse cursor for each mode insures that the designer always knows what the current mode is.

In Build mode, the static parts of the interface are drawn. For example, the designer might add to the window some widgets that should always be visible. Lines could be added as decorations. Many applications contain palettes that show which objects can be created, or that show various values for a property (like color, line-style, etc.). These palettes are drawn with Marquise in Build mode. In Run mode, the interface can be exercised to see how it will operate for the end user.

In Train mode, the designer operates the mouse and keyboard in the same way the end user would, and then goes into Show mode to demonstrate what the system's response should be. While in Train mode, the end-user behaviors are operational, but in addition, the keystrokes and mouse movements are saved. In Show mode, the designer can create and edit objects exactly the same as in

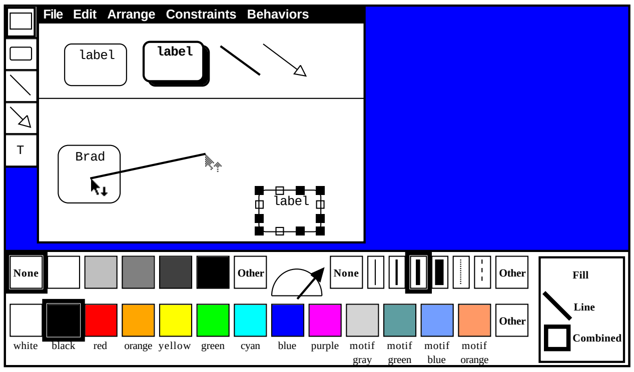

Figure 1:
The main Marquise windows: the basic object palette on the left, the main work area, and the palette for controlling the color and halftones for filling-styles and line-styles at the bottom. The designer is creating an interface with a ''create palette'' at the top containing two types of nodes and two types of links. The node at the lower right of the work window is selected. The Marquise commands are in the menubar at the top. The ''Constraints'' menu allows graphical constraints to be specified. The ''Behaviors'' menu allows objects to be declared as palettes, and displays the mode and feedback windows.

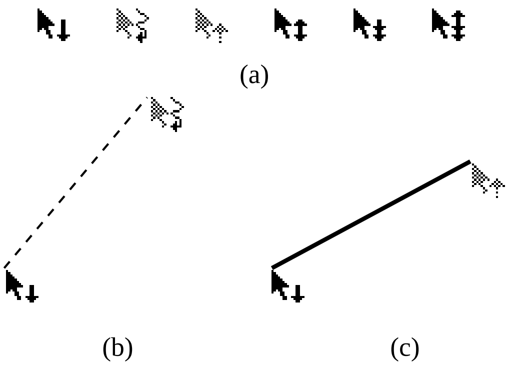

Figure 2:
(a) The icons that show where the mouse was pressed, moved to, released, clicked (pressed and released in the same place), double-clicked, and double clicked and released. (b) In Train mode, the designer pressed the mouse down and moved, and then in Show mode, drew a dotted line as the interim feedback. (c) Going back to Train mode, the designer released the mouse button, and in Show mode, deleted the dotted line and drew a solid line.

Build mode, except that the operations are remembered so they can be attached to the events demonstrated in Train mode.

As an example, here is how the designer would demonstrate that when the mouse button goes down, a feedback dotted line should be drawn which follows the mouse, and then when the button is released, the dotted line should be erased and a real line drawn. First, the designer would go into Train mode, press down the mouse button,

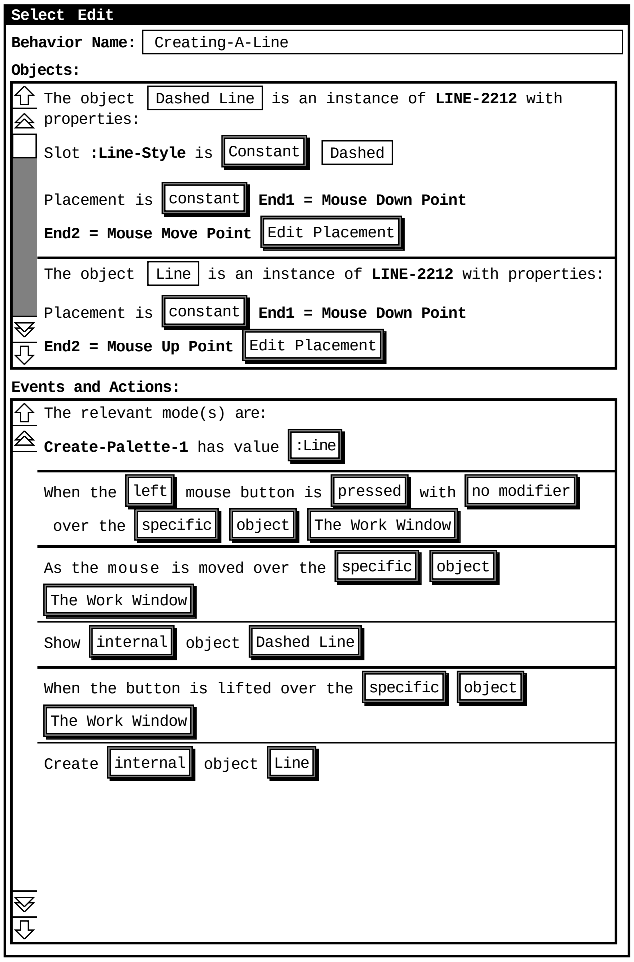

Figure 3:
The feedback window for behaviors. At the top is a pull-down menu of commands, then the name of the behavior, then the objects that participate in the behavior, and finally the events and actions. Pushing on the buttons displays a popup window of the other possible choices. Changing the option at the beginning of a ''sentence'' will change the options available for the rest of the sentence. An entire section of the window can be selected and cut, copied, etc.

and move away from the initial press. Without releasing the mouse button, the designer would change to Show mode. The window will now contain two icons which show where the mouse was pressed and to where it was moved. Now in Show mode, the designer draws a dotted line between the icons (Figure 2-b). Marquise infers that a dotted line should be created on the down press and one end should follow the mouse as it moves. Then the designer presses the mouse button somewhere on the background and switches to Train mode with the mouse button still down, so the mouse *release* can be demonstrated. Because this second demonstration does not include a down-press, the original down-press icon is retained. Next, in Show mode, the designer deletes the dotted line and draws a solid line from the initial down press icon to the final button release icon (Figure 2-c). This entire behavior takes less than 30 seconds to demonstrate, and very few new concepts or commands are necessary, since the designer already knows how to draw and delete objects in the editor.

If the mouse had been pressed and released in the same place, then a click icon would be displayed instead of the down, move and up icons. Double-clicking or double-clicking followed by a move are also supported. To allow modes to be changed while mouse buttons are being held down and while the mouse is at a particular place, keyboard accelerator keys are used to change modes.

Marquise generalizes from the designer's example actions to create the user interface. Any system that tries to generalize will sometimes guess wrong. Various mechanisms have been explored in other systems to show the user what has been guessed, so that users can verify and correct the inferences. Older systems, such as Peridot [3] and Druid [8], required the user to confirm each inference, which can be disrupting and annoying. In Marquise, a feedback window (Figure 3) shows the inferred operation. The labels and buttons can be read as a sentence, and the buttons can be pressed to pop up a list of other alternatives and change the values. Since Marquise appears to guess correctly most of the time, Marquise applies the inferred property immediately, and allows the designer to verify or change it afterwards in the feedback window.

ENVIRONMENT

Marquise makes heavy use of many features of Garnet. Garnet uses a retained object model and a prototype-instance object system. This means that there is an object in memory for every object on the screen, and that any object can be used as the prototype to make a copy or instance. Since all Garnet objects are represented the same way, there is a single mechanism for copying and creating objects, whether they are simple rectangles or aggregates containing many components. Therefore, Marquise can always generate appropriate code to create items for run time, without having to know the types of the objects the designer has drawn.

Implementing the behaviors that are demonstrated is quite straightforward once they have been determined because Marquise can create instances of ''interactor'' objects [5] and fill in the appropriate attributes. Each interactor implements a particular kind of behavior, such as selection, creation, moving, etc., and contains attributes to support most of the popular interaction styles.

The object system supports *constraints*, which are relationships that are declared once and maintained by the system. Constraints are used to maintain the relationships among the graphical objects in Marquise. Constraints can also be used to connect application data to Marquise-generated interfaces, or else application-specific call-back procedures can be invoked when a behavior is completed.

Garnet contains a number of other high-level interactive tools, such as the Lapidary tool for creating individual widgets or objects, the Gilt tool for editing dialog boxes, the Jade tool for automatically creating dialog boxes, and the C32 spreadsheet system for specifying complex constraints. Because all the tools use the same data structures and file format for describing objects, Marquise does not have to re-implement the functionality already provided by those tools—it can concentrate on the global behavior. The designer can have more than one tool operating at the same time, and use whichever is appropriate for the current part of the task.

FRAMEWORK

Marquise is able to construct the interface from the demonstrations because it has built-in knowledge of the kinds of operations that are common in graphical editors. This knowledge is part of the underlying Marquise framework that supports the interactive front end. The operations include: creating an object of the type in a palette, selecting objects, directly manipulating the size and shape with the mouse, specifying properties of objects (color, font, etc.) with a palette or property sheet, miscellaneous editing operations (deleting, duplicating, etc.), and application-specific commands.

Modes

It is very common in user interfaces for different behaviors to result from the same action, determined either by the location of the initiating event or by the value of a global mode variable. As an example of the first case, in MacProject II for the Macintosh, pressing the mouse button down will start text editing (if inside a box), select a box (if at its edge), create a new box (if in the background), draw a link between two boxes (if pressed in one box and released in another), grow a box (if pressed on a selection handle), or draw a link and create a box (if pressed in one box and released outside). Designers specify these differences in Marquise by demonstrating the different operations. Marquise notices what objects are underneath the demonstrated events (including the mouse release), so it can distinguish the correct times to use the different behaviors. The object being used is shown in the feedback window, and the buttons there can be used to edit the choice.

In other cases, the selection of the behavior is determined by the value of a global mode variable, which is set by a palette or an external application program. To make these modes explicit and visible, Marquise provides a *mode window*, shown in Figure 4, which lists each mode variable and its current value. The values displayed will change as the interface is operated, and the designer can directly edit the values for user modes. When the value has a fixed list of choices, these are available in a pull-down menu. To make an interaction dependent on whether a mode has the current value, it is only necessary to click on the check box next to the mode name before demonstrating the behavior. When a user action causes a mode to change, this can be demonstrated by simply editing the value in the mode window while in Show mode.

The combination of the icons and mode window make the control of the behaviors explicit and direct. In contrast, DEMO II [1] uses multiple examples to determine in which situations the operations should occur, which we feel will be more prone to errors.

Palettes

One of the important innovations in the Marquise framework is the formalization of a *palette*, which is a list of options, usually presented graphically. Each palette controls a single value or mode. Since palettes are conventional interaction techniques internally (i.e., they are usually a list of buttons), their internal behavior (how the user changes the current selection and what feedback shows the selected objects) can be easily specified using Lapidary or Marquise. The innovation here is the automatic connection of the palette to the rest of the interface.

Marquise identifies two main classes of palettes: *create* palettes and *property* palettes. A create palette contains the different kinds of objects that can be created. For example, the create palette for MacDraw II contains a selection arrow, strings, lines, rectangles, rounded-rectangles, ovals, arcs, curves, polygons, and text fields. A create palette for a CAD/CAM program for circuit boards might have a long list of different IC types, plus wires, pads, etc.

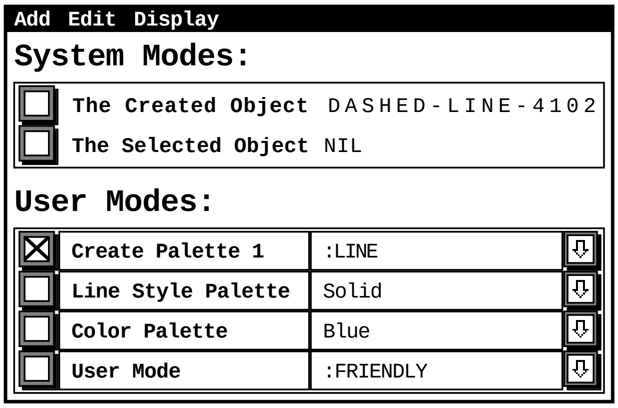

Figure 4:
The mode window showing the defined modes and their current values. The designer can click on the check box at the left of a row to indicate that the next action depends on the mode having the specified value. Modes based on palettes change as the palette is operated. Applications can also directly set a mode, and one of the actions resulting from a behavior might be a mode change.

A property palette contains the different values for a single property. MacDraw II has a property palette for the filling style at the top of the window, and property palettes for the line style, font size, font style, etc. in pull-down menus. Marquise supports palettes which are not always visible, and a palette can even be a subset of the items in a different widget (e.g., a section of a pull-down menu). In addition, the list of choices can be dynamically changing, for example, if the application has commands to read new libraries or to edit the palette itself.

There are two important distinctions between the two types of palettes. First, the create palette usually enables different interaction techniques. For example, the selection arrow enables selection, the rectangle enables dragging out new rectangles, and the text string enables clicking to start entering text. A property palette is assumed to only set values of properties and not to control interaction techniques.[2] Note that a create palette can enable various kinds of behaviors, such as selecting and deleting, and not just creating. The second difference between the two types of palettes is that Marquise assumes that objects cannot change type, so that selections in the create palette cannot affect the selected objects. However, clicking in property palettes usually changes the value of that property for the selected objects.

Create Palettes. To make a create palette, the designer only needs to draw the set of objects using Marquise or Lapidary, select them, and declare them to be a create palette using a menu command. Marquise will then add a row to the mode window (Figure 4) showing the palette and its current value. The designer would select the new row in the mode window, click on each item in the palette to put the system into the appropriate mode, and demonstrate the desired behavior.

The create palette has some additional attributes which control common features found in graphical editors. Some of these can be demonstrated, and the rest are specified in dialog boxes.

- In some palettes, when the user clicks on an item, that sets up a mode so that the next operation will create the kind of object represented by that item. This was shown in the example above, and is the way that MacDraw works. In other cases, clicking in the palette causes the object to be created immediately (e.g., at the current mouse position for a popup menu of choices, or at a computed place if the objects are laid out automatically by the system). Other times, objects are dragged off the palette.
- After an object is created, some applications select the newly-created object, some leave the selection unchanged, others add the new object to the selection set (if objects were selected before the create), and yet others clear the selection.
- Sometimes, after the object is created, the mode of the create palette will change. For example, in MacDraw II, after creating a rectangle, the mode changes back to selection (the arrow). However, if you double-click on the palette, the mode does not change after creation. In the original MacDraw, all object creation modes changed back to selection except for text strings.

Property Palettes. Property palettes allow the user to control the value of properties of objects. Typically, the same palette is used for specifying the global default value used for newly-created objects, and for changing the property of selected objects. The same palette might also be used to *show* the value for the selected object.

To create a property palette, the designer only needs to draw a set of objects representing the different values (for example, the line-style items of Figure 5), and declare them to be a property palette. Marquise then checks whether a single property seems to change in each element (as it does in Figure 5 and in most graphical property palettes), and if so, proposes this as the property to use. Alternatively, the user can specify the name of the property and the value for each item of the palette.

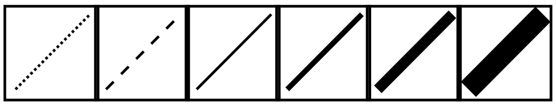

Figure 5:
After drawing this picture, the designer would select the lines, and declare them to be a property palette. Marquise would notice that the line-style changes, and would create an appropriate palette description.

Each primitive Garnet object describes which properties are relevant to it, and the designer can add additional properties for application-specific objects. Therefore, Marquise can automatically guess which properties are probably relevant to each type of object that is created. These guesses are reflected in the feedback and mode windows (Figures 3 and 4), and if Marquise guesses wrong, then the designer can adjust the values.

Some attributes for property palettes provided by Marquise are:

- Whether setting the property of a particular (selected) object also changes the global default used when a new object is created.
- If an object is selected that does not have the property represented by the palette (e.g., if the palette is for the font property and a line is selected), whether the palette goes inactive (greyed out) or not. When there are multiple objects selected, whether the palette is valid if at least one of the objects has this property, or only when all of the objects have this property.
- When an object is selected, whether the palette shows the value of that object. If more than one object is selected, then the palette might show the value only if all objects have the same value, pick the value of one of the objects to show, show the global default value, or just be cleared to show no value.
- If the palette does not echo the value of the property when the selection changes, then the newly selected object might get the current value of the property (as opposed to requiring another click in the palette after changing the selection).

Positions of new objects

Once Marquise knows which object to create, there is then

[2] This restriction could be lifted in the future if it becomes onerous, but it is consistent with the behavior of all editors we have studied.

the question of where and how to create it. There are two possibilities: the position is computed automatically, or is specified by the user with the mouse.

Automatic Layout. Garnet has built-in routines for list, table, tree, and graph layout. These automatically place the nodes, rather than requiring the user to specify a location. Each type of layout has a set of methods for creating and deleting nodes, and Marquise allows the designer to demonstrate how these methods are invoked and how their attributes are specified. Many previous systems have allowed a designer to build custom graph layout applications by writing code, but Marquise is the first to allow the look of the nodes to be drawn and the editing behaviors (creating, deleting, editing labels, etc.) to be demonstrated interactively.

First, the designer specifies which kind of layout is desired.[3] Next, the designer draws pictures to show the graphics for the nodes (and the graphics for the arcs for trees and graphs). If these have complex internal structure, then the Lapidary tool will be useful for drawing them. The built-in layout algorithms have many attributes that control the display, and some of these can be demonstrated (e.g., the spacing and direction). The rest are specified in a dialog box.

Next, the designer demonstrates the creation behavior. Using knowledge of the type of layout in use, Marquise tries to determine if the new object should be placed in some relation to a selected object, or globally with respect to all objects. For example, in a directed-graph editor, there might be commands for ‘‘Add new child’’ and ‘‘Add new parent.’’ Marquise does not try to understand the words in the command names. Instead, the designer would go into Run mode and select a node, and then in Train mode the designer would select the command. Finally, in Show mode, the new object would be created with the correct relationship to the selected node.

In some cases, the new object's position will not depend on the selection, but rather on global properties. For example, the new object might always go at the end of a list. In this case, the designer would make sure that no objects are selected before demonstrating the position of the new object, and Marquise would try to determine the appropriate place for the object. Alternatively, the position might depend on some global mode, so the appropriate row of the mode window would be selected before the demonstration. Usually, the position will be obvious (e.g., first or last), but if Marquise cannot guess it, then currently the designer will have to write a Lisp function to compute the position, possibly based on values in the mode window.

User Layout. Most graphical editors, however, require the user to explicitly specify the position of new objects. The example of Figure 2 shows how the simple case of a new line can be demonstrated in Marquise.

It is very common for the objects to be constrained in their placement. Marquise has built-in knowledge about gridding, so this can be easily used in an application. A more interesting problem is attachment. For example, an arrow connecting the boxes in Figure 6 might always be attached to the centers of the boxes. In an earlier article [4], we discussed how Lapidary allows the arrow prototype to be defined interactively with parameters that refer to the objects to which it should be attached. Lapidary creates constraints that keep the arrows attached as the objects move. Marquise allows the designer to interactively show how those parameters are filled in based on the designer's actions. For example, look back at Figure 1 where a creation palette is being drawn. To demonstrate the arrow creation mode, the designer would select the arrow in the create palette while in Run mode. This will change the value shown in the mode window for the create palette mode (Figure 4). The designer would then click on the check box next to this mode, which tells Marquise that the mode is significant for the next operation.

Assume that the arrow was defined so that setting the *from* and *to* parameters with objects would cause the line to be attached to those objects. In Train mode, the designer would press down inside a rounded-rectangle, and drag outside. Then, in Show mode, the designer would create an instance of the arrow with the shaft end inside the rounded-rectangle and the arrow end at the  icon. Then, the designer specifies that the rounded-rectangle corresponds to the *from* parameter, and Marquise infers that it should determine the parameter value based on where the mouse is first depressed, and that the other end should follow the mouse. Next, the designer demonstrates the mouse button-up response by deleting the feedback line and creating a new arrow between the two nodes. These nodes are declared as the *from* and *to* parameters. In the future, we will provide facilities for *gravity* so the designer could specify that while the mouse is moving, the feedback should jump to the attachment points of objects if they are close enough.

Selection

One of the most important operations in a graphical editor is selecting objects. Typically, the selected object will be shown by changing its appearance (e.g., to reverse video) or by showing ‘‘selection handles’’ around it (Figure 6). Marquise supports virtually any graphical response to show the selection. The designer simply draws an example of the selection graphics (or if the object itself changes, the designer draws the object first in its normal and then in its selected state). If the standard Garnet selection widget is desired, then it is only necessary to go into Show mode and select an object. A special line of the mode window shows which objects are selected, and this value can be edited to show whether the interaction being demonstrated adds to the selection set, removes from it, clears it, etc. This provides a uniform, intuitive mechanism for specifying almost any selection behavior. The designer can also specify

[3]Marquise cannot infer a new layout algorithm. For example, if a new kind of graph layout is required, the designer has to program it in Lisp, but it can then be used by Marquise-generated programs.

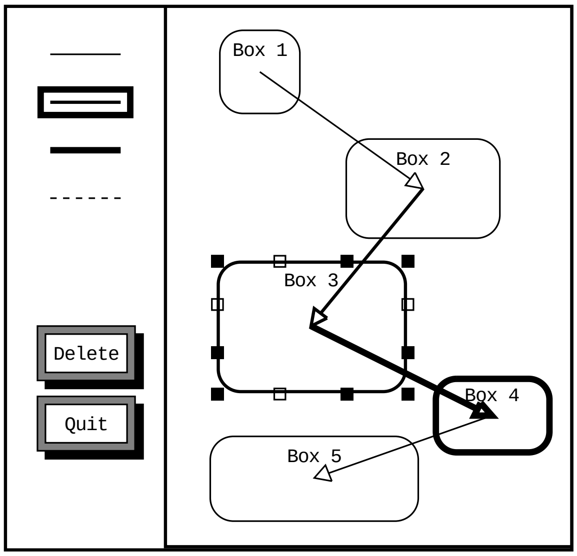

Figure 6:
The arrows are constrained to be in the centers of the boxes. Box 3 has ‘‘selection handles’’ around it, which show that it is selected, and the user can click on white handles to move it or black handles to grow it. The formula that computes the labels was hand-coded using C32.

whether a different form of feedback is used when there are multiple selections (as in Macintosh PowerPoint and MacProject II).

Moving and Growing Objects

Demonstrating what commands cause objects to be moved and grown works similarly to demonstrating how they are created: first, the designer demonstrates in Train mode what user action causes the interaction to start, and then in Show mode, moves or grows the appropriate object. Since the standard editing actions work in Show mode, the designer would just use the Marquise move-grow selection handles to demonstrate the behavior. Of course, if other objects are attached to the moved object with constraints, they will also move.

One complication is that often the object that the mouse is over is not the object that should be modified. For example, with selection handles, the user clicks on a handle, but wants to grow the object *underneath*. Marquise knows about this special case, and if the object the designer moves is attached by a constraint to the object clicked on, then this is reflected in the generated behavior.

Other Properties of Objects

Many properties of objects are controlled by palettes, but some are not. In some graphical editors, a menu command or double-clicking on an object opens a property sheet or dialog box with other properties. Marquise provides hooks to pop up a property sheet or a dialog box created automatically by Jade or interactively using Gilt. Of course, the designer can specify which fields are presented.

Miscellaneous Editing Commands

Because Garnet uses a retained object model, there is a standard format for all Garnet objects. Therefore, common editing commands such as bringing objects to the top (uncovered), sending to the bottom, cutting, copying, pasting, deleting (clear), duplicating, and printing in PostScript, are all provided. The designer simply demonstrates what action causes it to occur, and then which operation is desired. Note that unlike other frameworks that provide messages that must be overridden by each application, the code provided by Marquise for these operations can often be used without change.

Semantic Actions

Naturally, many of the commands in a graphical editor will invoke application-specific functions (sometimes called ‘‘semantic actions’’). Since these may involve arbitrary computation, it is impossible for Marquise to infer these from a demonstration. However, techniques like those previously reported for Gilt [7] are used to allow the application procedures to be independent of the way they are invoked (from a button, menu, double-click, etc.) and somewhat independent of the graphics. However, most functions will want to walk through the graphical objects computing values, so they will clearly have to look at the graphical objects in the window.

If the result of the function is a change to the graphic appearance of nodes, then this can be specified demonstrationally. For example, a ‘‘critical-path’’ command in a graph editor might want all the nodes on the critical path to turn red. The designer can bring up a property sheet on the nodes, add an `on-critical-path` property,[4] and demonstrate that the nodes are black when it is NIL and red when it is T. Then, the critical-path function would only be responsible for setting the `on-critical-path` value in each node. This makes the application function more independent of the graphical response to its actions.

Semantic feedback can often be provided in the same way. For example, Marquise supports highlighting of only those objects that an object is being dragged can legally be dropped into, as in the Macintosh Finder. Here, a function could be called to set a particular property of each object to T or NIL. Then, the designer would demonstrate the appropriate color change when the node is over an object which has the value T for that property.

Similarly, if the application wants to control which mode is in effect, it can simply change the value of one of the mode variables, and the designer can demonstrate interactively what this controls.

EDITING

An important aspect of an interactive builder is how to edit the interfaces after they have been created. It is easy to edit

[4]The Garnet object system allows properties to be added to objects at any time.

the graphics, since they can be directly manipulated in Build mode. For the behaviors, the feedback window of Figure 3 shows the properties. When in Train mode, the feedback window continually shows the name and properties of the behaviors being executed, so the designer can determine which behaviors are associated with which events. There are also commands to list all the behaviors, or all those affecting a particular object.

CONCLUSION

One of the important questions for an interactive tool is what is the range of interfaces that it can create. Unfortunately, this is very difficult to quantify, except by example. Using the Lapidary, Gilt and Marquise tools in Garnet, it is possible without programming to create complete user interfaces like those in Macintosh MacDraw, MacDraw II, PowerPoint, and MacProject II (which are surprisingly different), as well as applications with various kinds of automatic layout for nodes. Later, we hope to expand the range of Marquise to handle gestural interfaces (the Garnet toolkit already supports gesture recognition), and those with 3-D graphics. We also plan to add support for animations, which will probably make possible the demonstration of various visualizations and video games. Another addition will be to support defining constraints among objects directly in Marquise, probably using demonstrational techniques similar to Peridot [3] or Druid [8].

Marquise is still under development. When it is more robust, we will perform user-testing to see if the demonstrations and feedback are understandable to both non-programmers and programmers. After that, we will release it for general use as part of the Garnet system. All this will help show what kinds of behaviors it can capture, and we will continually work to expand the range.

We believe that interactive, demonstrational creation of user interfaces is easier, faster, and more fun than programming. Many interactive builders have already shown that dialog boxes and forms can be created interactively. Marquise shows that direct manipulation techniques can be used to generate the user interfaces of a much wider class of graphical applications as well.

ACKNOWLEDGEMENTS

For help with this paper, we would like to thank Dario Giuse, Brad Vander Zanden, Andrew Werth, and Bernita Myers.

This research was sponsored by the Avionics Laboratory, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U. S. Air Force, Wright-Patterson AFB, OH 45433-6543 under Contract F33615-90-C-1465, ARPA Order No. 7597.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

REFERENCES

1. Gene L. Fisher, Dale E. Busse, and David A. Wolber. Adding Rule-Based Reasoning to a Demonstrational Interface Builder. ACM SIGGRAPH Symposium on User Interface Software and Technology, Proceedings UIST'92, Monterey, CA, Nov., 1992, pp. 89-97.

2. Anthony Karrer and Walt Scacchi. Requirements for an Extensible Object-Oriented Tree/Graph Editor. ACM SIGGRAPH Symposium on User Interface Software and Technology, Proceedings UIST'90, Snowbird, Utah, Oct., 1990, pp. 84-91.

3. Brad A. Myers. *Creating User Interfaces by Demonstration.* Academic Press, Boston, 1988.

4. Brad A. Myers, Brad Vander Zanden, and Roger B. Dannenberg. Creating Graphical Interactive Application Objects by Demonstration. ACM SIGGRAPH Symposium on User Interface Software and Technology, Proceedings UIST'89, Williamsburg, VA, Nov., 1989, pp. 95-104.

5. Brad A. Myers. Encapsulating Interactive Behaviors. Human Factors in Computing Systems, Proceedings SIGCHI'89, Austin, TX, April, 1989, pp. 319-324.

6. Brad A. Myers, Dario A. Giuse, Roger B. Dannenberg, Brad Vander Zanden, David S. Kosbie, Edward Pervin, Andrew Mickish, and Philippe Marchal. "Garnet: Comprehensive Support for Graphical, Highly-Interactive User Interfaces". *IEEE Computer 23*, 11 (Nov. 1990), 71-85.

7. Brad A. Myers. Separating Application Code from Toolkits: Eliminating the Spaghetti of Call-Backs. ACM SIGGRAPH Symposium on User Interface Software and Technology, Proceedings UIST'91, Hilton Head, SC, Nov., 1991, pp. 211-220.

8. Gurminder Singh, Chun Hong Kok, and Teng Ye Ngan. Druid: A System for Demonstrational Rapid User Interface Development. ACM SIGGRAPH Symposium on User Interface Software and Technology, Proceedings UIST'90, Snowbird, Utah, Oct., 1990, pp. 167-177.

9. John M. Vlissides and Mark A. Linton. Unidraw: A Framework for Building Domain-Specific Editors. ACM SIGGRAPH Symposium on User Interface Software and Technology, Proceedings UIST'89, Williamsburg, VA, Nov., 1989, pp. 158-167.

10. David Wolber and Gene Fisher. A Demonstrational Technique for Developing Interfaces with Dynamically Created Objects. ACM SIGGRAPH Symposium on User Interface Software and Technology, Proceedings UIST'91, Hilton Head, SC, Nov., 1991, pp. 221-230.